Design Methods For Real-Time Systems In Ada

Kevin L. Mills

CS 619

July 21, 1993

Design Methods For Real-Time Systems In Ada

Whether in commercial or military applications, no real-time system can simply be coded straight-away from the requirements. A method is needed to enable system designers to move from a requirements statement through a decomposition of the problem, to a system architecture, and then to an identification and specification of system components. For real-time systems, a design method must help identify concurrency and shared access to resources. The products of a design method should enable low-level design and coding assignments to be distributed to programmers. In many cases, the programmers will be required to code in Ada[1], a language developed with concurrent, real-time systems in mind.

Several methods exist for designing real-time systems that will be implemented in Ada. This paper helps a designer to choose between two such design methods. One method, developed by Nielsen and Shumate at the Hughes Aircraft Company, aims specifically at large, real-time systems that will be implemented in Ada.[2,3] The second method, devised by Gomaa, applies more generally to real-time systems, but includes some optional steps when the implementation language is Ada.[4,5,6] For each method, this paper compares the goals, the underlying assumptions, the design steps, and the resulting products. Then, the two design methods are evaluated , and one of the methods is recommended for designing real-time control systems that will be implemented with Ada. First, the Nielsen and Shumate method is reviewed.

Designing Large Real-Time Systems With Ada

Nielsen and Shumate have devised a method for designing large, real-time systems with Ada concepts and constructs. Nielsen and Shumate argue that:

> Ada requires a new design method for real-time systems since it differs from conventional language/executive combinations in two important ways: (1) concurrency and process communication are inherent in the language and do not require a separate RTE [Real-Time Executive]; and (2) the nature of the tasking model is different from current popular approaches used for task scheduling, communication, and synchronization.[3, p. 696]

Nielsen and Shumate also compare their approach to a few previous design methods specifically intended for real-time systems. A summary of the Nielsen and Shumate design method is presented in Table 1.

As with most design methods for real-time systems, Nielsen and Shumate begin by defining the boundaries between hardware and software. This definition is documented in a context diagram. Knowing that each hardware device will require a controlling task, Nielsen and Shumate immediately assign processes to the edges of the context diagram to interact with each device. This step is documented in a preliminary concurrent process graph.[1] This graph is preliminary in the sense that only some of the concurrent processes that will ultimately be identified are shown. All that remains is to decompose the middle part, i.e., that portion of the software that does not control directly the hardware.

To decompose the middle part, Nielsen and Shumate incorporate structured analysis to produce data flow diagrams (DFDs) that divide the system into a set of functions and the data and control flows

---

[1] Readers interested in examples of a process graph and other products discussed here and in the presentation of ADARTS should consult the references. The current paper evaluates the two methods but does not include a tutorial on the methods themselves. Only the briefest overview necessary to support the evaluation and recommendations is given.

Table 1. Summary Of Nielsen And Shumate Design Method

Goal: Provide a consistent design method for Ada implementations of real-time systems.	**Assumptions:** Underlying implementation to be built on Ada.
Design Method Steps	**Design Method Products**
1. Determine Hardware Interface	1. Context Diagram
2. Assign Processes To Edge Functions.	2. Preliminary Concurrent Process Graph
3. Decompose Middle Part.	3. Data Flow Diagrams.
4. Determine Concurrency.	4. Process Identification Diagram.
5. Determine Process Interfaces.	5. Process Structure Chart.
6. Introduce Intermediary Processes.	6. Ada Task Graph.
7. Encapsulate Tasks In Ada Packages.	7. Ada Package Graph.
8. Translate Design Into Ada PDL.	8. Ada Package Specifications.
9. Decompose Large Tasks.	9. Partial Ada Package Bodies.
10. Conduct Design Reviews.	10. Updated Documentation.

between them. Nielsen and Shumate then advise synthesizing functions from the DFDs into concurrent processes. This step is guided by a set of "process selection rules", or heuristics, that include: identifying external devices (already applied), applying functional cohesion, identifying time-critical functions, identifying periodic functions, identifying low-priority (but computationally intensive) functions, applying temporal cohesion, identifying memory constraints, and identifying shared-access databases. The result of this step is a process identification chart -- more simply, the DFDs with rectangular boxes drawn around those transforms that are grouped together as a concurrent process.

At this point, Nielsen and Shumate transfer the identified processes onto a new sheet of paper and then consider the interfaces between them. The connections between transforms (now grouped into processes) on the DFDs provide a starting point. Data stores and their access arcs are copied directly from the DFDs. Events on the DFDs are translated into signals on the new sheet. Data flows to and from devices are copied directly from the DFDs. This leaves only the data flows between processes to consider. Here, the main decision is between loosely-coupled message exchange (through queues) and tightly-coupled message exchanges (i.e., send a message and wait for a reply or send a message and wait for the message to be accepted by the receiving task). At the end of these deliberations a process structure chart is produced. The next step is to consider any intermediary processes that may be required.

Intermediary processes are necessary to effect queues between tasks and to control access among tasks to shared data stores. In a sense, intermediary tasks are needed to overcome some of the limitations of the Ada inter-task communications paradigm. (For example, Ada does not directly support loosely-coupled message exchange or mutually-exclusive, shared access to encapsulated data structures.) To correctly perform this step, a designer must understand the features of Ada task synchronization. This step ends once an Ada task graph is produced to show every Ada task in the design and each Ada communications mechanism between tasks.

Now the Ada tasks are encapsulated in Ada packages. Nielsen and Shumate group tasks from the Ada task graph into packages using a set of heuristic packaging rules that include: encapsulation guidelines, functionality, reusability, coupling, visibility, dependency, recompilation, object-orientation, and data store protection. The result of this step consists of Ada package graphs.

Using the Ada package graphs, Nielsen and Shumate translate the design into Ada program design language (PDL). This step in practice means producing Ada package specifications and partial bodies, while making liberal use of the Ada **separate** keyword to defer description of algorithms and other design details. Comments are inserted to indicate when a task calls other tasks. Using Ada as the PDL lets the designer rely on the compiler to check for consistency and also lays out the Ada task interfaces in detail that applies directly to subsequent coding. The result of this step includes Ada package specifications and partial package bodies.

From here, any large tasks are further decomposed into subunits. These subunits are also specified using Ada. The results of this step may include additional package specifications and partial package bodies. The final step of the Nielsen and Shumate design method encompasses a preliminary and critical design review; after each review the design and related documentation are revised as necessary.

The bulk of the notation used in the Nielsen and Shumate method should be familiar to readers who understand DFDs. At some points, the design method uses the bubbles to represent functions and at other points to represent processes or tasks -- the purpose is clear from the context. Where inter-process communications are illustrated, the notation used is adapted from Gomaa's DARTS method.[4,6] The remaining notation in the Nielsen and Shumate method is Ada. An alternative to the Nielsen and Shumate method is an Ada-based Design Approach for Real-Time Systems (ADARTS) as proposed by Gomaa.

An Ada-based Design Approach for Real-Time Systems (ADARTS)

ADARTS is a member of a family of design methods for real-time systems developed over the past decade by Gomaa. The original method was a Design Approach for Real-Time Systems (DARTS).[4,6] DARTS served as the basis for ADARTS and also influenced the thinking of Nielsen and Shumate. Subsequent to ADARTS, Gomaa proposed an extension, called DARTS/DA, for designing distributed, real-time applications.[7] In what follows, only the specific ADARTS design method is considered.[5]

Gomaa developed ADARTS to extend the concurrent tasking aspects of DARTS to include an emphasis on information hiding:

> In addition to supporting mapping to Ada, a goal of this extension was to provide more maintainable and reusable designs by adopting a significantly greater use of information hiding than DARTS. [...] An alternative approach to Real-Time Structure Analysis is provided for analyzing and modeling the system under development, namely Concurrent Object-Based Real-time Analysis (COBRA). This emphasizes the decomposition of a system into subsystems that provide a set of services supported by objects and functions.[5, p. 11-2]

While ADARTS includes a front-end analysis step that encompasses either Real-Time Structure Analysis (RTSA) or COBRA, we will consider only the use of COBRA. These two approaches are similar; however, COBRA employs more advanced, object-based thinking than RTSA. (The reader who is unfamiliar with RTSA can consult Ward and Mellor to further explore this topic.[8]) A summary of the ADARTS design method is presented in Table 2.

Table 2. Summary Of ADARTS Design Method

Goal: Provide the decomposition principles and steps for structuring concurrent and real-time systems into concurrent tasks and information hiding modules.	**Assumptions:** Design will not necessarily, but may be, implemented in Ada.
Design Method Steps	**Design Method Products**
1. Develop Environmental and Behavioral Model. (This is the COBRA or RTSA front-end analysis.)	1. a. Context Diagram b. State Transition Diagrams. c. Control and Data Flow Diagrams. d. Data Dictionary. e. Mini-specifications.
2. Structure System into Concurrent Tasks.	2. a. Task Architecture Diagram. b. Task Behavior Specifications.
3. Structure the System into Information Hiding Modules.	3. a. Information Hiding Module (IHM) Classification. b. Informal IHM Specifications. c. System Architecture Diagram.
4. Develop an Ada-based Architectural Design (required only for Ada implementation).	4. a. Ada Architecture Diagram. b. Updated Task Behavior Specifications.
5. Define Ada Component Interfaces (required only for Ada implementation).	5. Ada Package Specifications.

ADARTS begins with the expected Context Diagram depicting the boundary between hardware and software. If the system is decomposed into subsystems, a context diagram is provided for each subsystem. Where subsystems communicate, each appears as a terminator on the other's subsystem context diagram. COBRA provides some criteria for decomposing a system into subsystems and also gives examples of common types of subsystems: real-time control, real-time coordination, data collection, data analysis, server, aggregate object, user services, and system services.

For each subsystem, COBRA addresses the concern of identifying objects in the problem domain and then of identifying functions that interact with those objects. Objects and functions are then arranged into Control/Data Flow Diagrams (C/DFDs) that illustrate the relationships between the objects and functions in the subsystem. Since C/DFDs may be hierarchically structured, only the leaf node objects and functions are considered for potential concurrency. COBRA provides some criteria for identifying objects in a subsystem. Potential objects include: physical device I/O objects, control objects, data abstraction objects, algorithm objects, and user role objects. As functions are identified, the analyst must relate the functions to appropriate objects. Here, too, COBRA gives help. Asynchronous functions are triggered by an object or another function to perform an action. When an asynchronous function is state-dependent, then it must be triggered by a control

object. Periodic functions are activated at regular intervals to perform some action. When a periodic function is state-dependent, a control object will enable and disable it.

To stimulate an analyst's thinking, COBRA prescribes a technique called behavioral scenario analysis. This technique requires the analyst to identify each external event that can occur and to trace the operations through a subsystem once the event occurs. In this manner, the analyst can find overlooked objects, functions, and relationships. Using this technique the analyst can also build up state transition diagrams for each control object in the subsystem.

Once the COBRA process is completed, the designer will have produced several valuable products, all aimed at analyzing the problem. The system and subsystem context diagrams bound the software system and depict the relationships between the subsystems. For each subsystem, the C/DFDs identify the objects, functions, and data stores and the relationships between them. For every control object, a state transition diagram defines the behavior of the object. For every data store, a data dictionary entry defines the syntax and semantics of the information in the store. For every non-control object and for each function, a mini-specification describes the required behavior in structured English. The notation used in these products is the same as the notation used for RTSA, except that data transforms in COBRA can represent objects (labeled with nouns) as well as functions (labeled with verbs).

Once the behavioral model is completed, the designer can consider task structuring. To help the designer, ADARTS provides a set of task structuring criteria organized into four broad categories: 1) I/O Event Dependency, 2) Internal Event Dependency, 3) Task Cohesion Criteria, and 4) Task Priority Criteria. Within each category, detailed criteria are given. In general, the designer applies the categories of criteria sequentially in the order listed. Categories one and two are aimed at defining the greatest possible concurrency from the behavioral model. Since excessive concurrency can reduce system performance (by introducing undue context switching), the Task Cohesion Criteria are then applied to reduce the number of concurrent tasks. Finally, the Task Priority Criteria are used to identify the relative priority of each task. (ADARTS is weak in the area of task priority criteria.)

The result of ADARTS task structuring is a Task Architecture Diagram (TAD) which illustrates each task in the system, the communications mechanisms between the tasks, and the interaction of the tasks with the environment. For each task shown on the TAD, a Task Behavior Specification (TBS) is developed that details: 1) task inputs and outputs (as messages, events, and data), 2) criteria for structuring the task and reference to the C/DFD transforms that are included in the task, 3) task timing characteristics, 4) relative priority of the task, 5) an outline of the task's logic, and 6) any errors detected by the task.

Once the tasks are structured, ADARTS requires the designer to structure the system into information hiding modules (IHMs); of course, the designer is given criteria to help accomplish the necessary structuring. In general, ADARTS defines six types of IHMs: 1) device interface modules (hide the actual interfaces to hardware), 2) data abstraction modules (hide the internal structure of data stores), 3) state transition modules (hide the contents of state transition tables), 4) function driver modules (hide the rules for determining what should be output to a device), 5) algorithm hiding modules, and 6) software decision modules (hide a software decision considered likely to change). Each module identified is classified (by type) and then an informal specification is created to describe: 1) the information hidden, 2) the structuring criterion used to identify the module, 3) any assumptions about the module (e.g., multiple readers/writers), 4) anticipated changes, and 5) the operations provided by the module.

After the IHMs are identified and specified, the TBSs are updated to include, as necessary, specific references to IHMs and associated operations. Also, the TAD is updated to show the IHMs and their relationships to the tasks. The resulting product is called a System Architecture Diagram (SAD). IHMs can be encapsulated within tasks or shared between tasks. Those IHMs that are shared between tasks may exhibit requirements for shared-access control (as documented in the assumptions section of the IHM specifications).

Up to this point ADARTS has produced a rigorous analysis and design, independent of any particular language or environment. Under many circumstances, ADARTS terminates at this point; however, when the design will be implemented in Ada, two additional steps are provided. The first additional step involves identifying any required Ada support tasks (these were called intermediary tasks by Nielsen and Shumate). ADARTS attempts to limit the proliferation of Ada support tasks to those required to implement loosely-coupled message queues and those needed to synchronize access to shared data. Once the support tasks are identified, ADARTS requires that they be encapsulated in Ada packages. The SAD is then updated to include the new Ada packages -- the resulting diagram is called an Ada Architecture Diagram. Next, informal Ada task and package specifications are produced and the TBSs are updated to include references to Ada packages. The second additional, Ada-specific step in ADARTS requires that formal specifications be written for each component interface. These specifications are described using Ada.

Evaluation Of The Two Design Methods

Now, each method will be evaluated against the following criteria: 1) rigor, 2) repeatability, 3) understandability, 4) effort required, 5) scope of applicability and 6) automated support. With regard to rigor, ADARTS is clearly superior. The use of finite state machines to define control behavior, the relationship of those finite state machines to the C/DFDs, the traceability provided between each of the products of the process, the use of behavioral scenario analysis to identify functions and relationships, all of these, present in ADARTS and absent in the Nielsen and Shumate method, increase the rigor of the design process. Nielsen and Shumate do provide rigor when defining the task and package specifications using Ada -- this rigor is missing from ADARTS unless the final extra step for Ada implementations is included.

ADARTS appears more repeatable. The steps are so well-defined, the criteria so detailed, and the products so precisely described that the method can be applied probably by different designers with similar results, and can also be applied by the same designer to different problems with similar resulting design quality. This is not to suggest that creative designers might not reach vastly different results that are of high quality, or that great designers might not produce great results using different design methods (or no design methods); rather, given an average set of competent designers, the ADARTS approach, being more detailed and systematic, should produce designs of acceptable quality with more certainty and with more consistency than the Nielsen and Shumate method.

Evaluating the understandability of the two methods is not easy. On the one hand, given a little familiarity with structured analysis, with concurrency, and with Ada, the Nielsen and Shumate method should prove easily understandable. Certainly, there is less to examine and consider. On the other hand, given little or no familiarity with the specific problem being solved, or with Ada, ADARTS provides a much more comprehensive, solution-independent analysis of the problem.

The effort required to produce an ADARTS design far exceeds that necessary to create a design using the Nielsen and Shumate approach. The quantity of documents produced, the interrelationships between those documents, and the lack of automated support (until one gets to the final step of specifying Ada packages) all contribute to demand more labor for an ADARTS design than for a comparable design using the Nielsen and Shumate method.

With regard to scope of applicability, the clear edge goes to ADARTS. This a natural consequence of the assumption by Nielsen and Shumate that designs developed using their method will be implemented in Ada. ADARTS provides a design method that remains independent of Ada until the final two, optional, steps. Nielsen and Shumate point out that their approach can be adapted should the final implementation not be coded in Ada. While this is certainly true, they do not show how to make such adaptation, nor does the path appear smooth. ADARTS also provides a clear delineation between problem analysis, solution design, and Ada design. This makes ADARTS more modular than the Nielsen and Shumate approach; such modularity can allow

different analysis techniques to be used (e.g., RTSA or COBRA) and can readily lead to different implementation approaches (e.g., Ada run-time environment or C code and a real-time executive).

While considering scope of applicability, largeness deserves special mention. Because their book and article carry the adjective large in the title, the reader naturally expects that one aspect of the Nielsen and Shumate design method that sets it apart from others is the facility to design *large*, real-time systems. This is not so. Apparently, *large* to Nielsen and Shumate means encumbered with bureaucracy. The chapter in their book that deals with programming-in-the-large allows them to intimidate the reader with buzz words and concepts, but does nothing to address the problem of designing *large*, real-time systems. ADARTS, on the other hand, quietly provides a systematic approach to decompose a *large*, real-time system into subsystems, to represent the coupling between subsystems, and to then address each subsystem independently in the design.

Although neither the ADARTS nor Nielsen and Shumate design methods are supported specifically by automated tools (unless RTSA is used as the front-end to ADARTS), the Nielsen and Shumate approach gets more quickly to a level where automated support is possible from an Ada compiler. ADARTS might never reach this state, unless Ada is the implementation language.

Recommendation

For a large, real-time system, ADARTS should be used as the design method in lieu of the Nielsen and Shumate approach. Why? ADARTS is more modular, more rigorous, more repeatable, and can be applied independent of the intended implementation environment. Since the system is large, investing more effort to document and maintain the design should not be a significant concern. Also, since ADARTS provides clearly separable analysis, design, and implementation-related products, the chance of reusing some of the results appears likely. Applying ADARTS might require more time before Ada package specifications are produced, but managers should have an increased confidence that the problem is well-understood, that the design is robust, and that coding and testing effort will not be wasted through unnecessary redesign.

Where the real-time system is small, the problem is well-understood, and the target environment is Ada, the Nielsen and Shumate design will lead to a reasonable solution more quickly and at lower cost than ADARTS. Even here, the Nielsen and Shumate approach could be improved by using RTSA to decompose the middle part and by using finite state machines to describe the behavior of RTSA control transforms. Of course, to define correctly the finite state machines, behavioral scenario analysis will be required. At this point, the Nielsen and Shumate design method would become very close to ADARTS, but would lack the rigorous, repeatable, and detailed criteria for task structuring and information hiding.

References

[1] United States Department of Defense, Reference Manual for the Ada programming language, ANSI/MIL-STD-1815A-1983, Department of Defense, Washington, D.C., January 1983.

[2] Kjell Nielsen and Ken Shumate, Designing Large Real-Time Systems With Ada, McGraw-Hill Book Company, New York, 1988, 464 pages.

[3] Kjell Nielsen and Ken Shumate, "Designing Large Real-Time Systems With Ada",

Communications of the ACM, August 1987, pp. 695-715.

[4] H. Gomaa, "A Software Design Method For Real-Time Systems", *Communications of the ACM*, September 1984, pp. 938-949.

[5] H. Gomaa, "A Design Approach for concurrent and Real-Time Systems (ADARTS)", in Course Notes On Software Design Methods - Part III, George Mason University, Fall Semester 1992, approximately 200 pages.

[6] H. Gomaa, "Software Development Of Real-Time Systems", *Communications of the ACM*, July 1986, pp. 657-668.

[7] H. Gomaa, "A Software Design Method For Distributed Real-Time Applications", Journal of Systems and Software, February, 1989, 9 pages.

[8] P. T. Ward and S. J. Mellor, Structure Development for Real-Time Systems, Yourdon Press, New York, Three Volumes, 1985-1986.